ISSN: 2395-1680 (ONLINE)
DOI: 10.21917/ijme.2018.0101

DESIGN AND IMPLEMENTATION OF CACHE MEMORY USING CMOS TRANSISTORS

G. Chinavenkateswararao, M. Kamaraju and P.V. Subbarao
Department of Electronics and Communication Engineering, Gudlavalleru Engineering College, India

Abstract

Cache systems are on-chip memory component utilized to store information. Cache serves as a buffer between a CPU and its principle memory. Cache memory is utilized to synchronize the data transfer rate between CPU and principle memory. As cache memory closer to the smaller scale processor, it is faster than the Random access memory (RAM) and principle memory. The advantage of storing data on cache, as compared to RAM, is that it has faster retrieval times, but it has disadvantage of on-chip energy consumption. In term of detecting number of cache hits and miss rate for the range of instructions and less power consumption, the efficient cache memory will be proposed using CMOS transistors. Evaluate the performance of cache memory in terms of power dissipation and speed of operations. Design architect IC, a tool of mentor graphics, is used for designing schematic diagram and eldonet is used for simulation of designed model.

Keywords:

Cache Memory, Gates, Flip Flops, Register Index, Tag and Power

1. INTRODUCTION

Originally, memory was as fast as CPUs. Over time CPUs got faster and faster. Memory got faster too, but didn't keep up with the increasing speed of CPUs, which means that CPUs needed to spend more and more time waiting for data to come from memory which can have a serious effect on performance. As a work-around for the speed difference between CPUs and memory, CPU manufacturers started including caches. The basic idea is to have an small area of memory called cache that the CPU can access really fast, that's used to store a copy of data in cache to reduce the time the CPU spends waiting to access main memory [1].

The cache memory is the memory which is very nearest to the CPU; all the recent instructions are stored into the cache memory. The cache memory is attached for storing the input which is given by the user and which is necessary for the CPU to perform a task. But the capacity of the cache memory is too low in compare to memory and hard disk [2].

The cache memory lies in the path between the processor and the memory. The cache memory therefore, has lesser access time than memory and is faster than the main memory. Cache memory have an access time of 100ns, while the main memory may have an access time of 700ns. The cache memory stores the program currently being executed or which may be executed within a short period of time. The cache memory also stores temporary data that the CPU may frequently require for manipulation [3].

The cache memory works according to various algorithms, which decide what information it has to store. These algorithms work out the probability to decide which data would be most frequently needed. This probability is worked out on the basis of past observations. Cache is made up of a group of registers. Each register has an instruction copy of the same instruction in some backing store (main memory). Each entry also has a tag, which specifies the identity of the instruction in the backing store of which the entry is a copy [4].

When the cache client needs to access an instruction presumed to exist in the backing store, it first checks the cache. If a register can be found with a tag matching that of the desired instruction, the instruction in the cache is used instead. This situation is known as a cache hit. The percentage of accesses in cache hits is known as the hit rate or hit ratio of the cache [5].

The alternative situation, when the cache is consulted and found not to contain an instruction with the desired tag, has become known as a cache miss. The previously un-cached instruction fetched from the main memory during miss handling is usually copied into the cache, ready for the next access [6].

More efficient caches compute use frequency against the size of the stored contents, as well as the latencies and throughputs for both the cache and the backing store. This works well for larger amounts of data, longer latencies and slower throughputs, such as experienced with a hard drive and the internet, but is not efficient for use with a CPU cache [7].

1.1 NEED FOR CACHE

The cache memory is due to the mismatch between the speeds of the main memory and the CPU. The CPU clock very fast, whereas the main memory access time is comparatively slower. Hence, no matter how fast the processor is, the processing speed depends more on the speed of the main memory. It is because of this reason that a cache memory having access time closer to the processor speed is introduced [8] [9].

It acts as a high speed buffer between CPU and main memory and is used to temporary store very active data and action during processing since the cache memory is faster than main memory, the processing speed is increased by making the data and instructions needed in current processing available in cache. The cache memory is very expensive and hence is limited in capacity [10].

The following sections will provide a brief overview of architecture of the cache memory and explanation about the implementation. Then brief description about the comparator, multiplexer, demultiplexer, data register and memory cell sections are given. The first section describes about the overall memory block diagram. Later the individual modules in the system explained. Finally few notes on simulation results.

2. CACHE MEMORY ARCHITECTURE

The proposed cache architecture mainly consists of register tag, comparator, multiplexer, demultiplexer, memory and data registers and its block diagram is shown in the Fig.1.

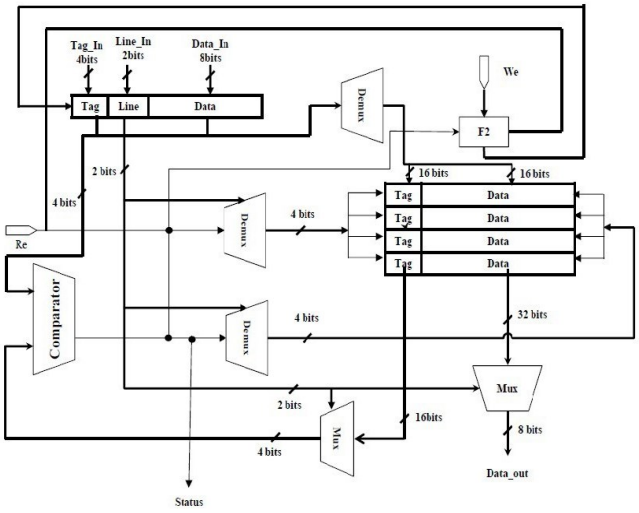

Fig.1. Cache Memory Architecture

Cache reads are the most common CPU operation that takes more than a single cycle. Program execution time tends to be very sensitive to the latency of a level-1 data cache hit. A great deal of design effort, and often power and silicon area are expended making the caches as fast as possible.

The simplest cache is a virtually indexed direct-mapped cache. The virtual address is calculated with an adder, the relevant portion of the address extracted and used to index an SRAM, which returns the loaded data. The data is byte aligned in a byte shifter, and from there is bypassed to the next operation. There is no need for any tag checking in the inner loop in fact, the tags need not even be read. Later in the pipeline, but before the load instruction is retired, the tag for the loaded data must be read, and checked against the virtual address to make sure there was a cache hit. On a miss, the cache is updated with the requested cache line and the pipeline is restarted.

A simple cache memory is shown on the above Fig.1. It has three main parts: a directory store, a data section, and status information. All three parts of the cache memory are present for each cache line. The cache must know where the information stored in a cache line originates from in main memory. It uses a directory store to hold the address identifying where the cache line was copied from main memory. The directory entry is known as a cache-tag.

A cache memory must also store the data read from main memory. This information is held in the data section (see Fig.1).

The size of a cache is defined as the actual code or data the cache can store from main memory. Not included in the cache size is the cache memory required to support cache-tags or status bits. The status bits in cache memory to maintain state information. Two common status bits are the valid bit and dirty bit. A valid bit marks a cache line as active, meaning it contains live data originally taken from main memory and is currently available to the value or specified indirectly as the contents of any of the eight registers.

2.1 FUNCTIONALITY OF DIRECTED MAPPED CACHE

Direct mapping is one method of deciding where blocks of memory will be stored in the cache. Each block of memory is assigned a specific line in the cache. Since the cache is smaller than the memory, multiple blocks will share a single line in the cache. If a line is already full when a new block needs to be written to it, an old block will be overwritten.

If a program continually accesses multiple blocks of data that share the same line in a direct mapping cache, the line will be rewritten often. This results in a lot of misses because the data the computer needs is less likely to be the data that is actually in that cache line at the moment. So, direct mapping has a lower hit rate than other cache mapping models. The Fig.2 illustrates direct memory mapping technique in a cache memory.

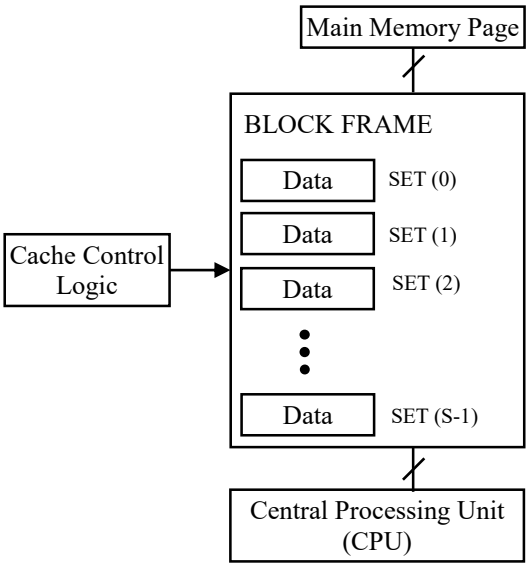

Fig.2. Block diagram of direct mapped cache Memory Architecture

The cache memory input buffer register is of 14-bit size of which 8-bit are data input bits, 6-bits are address bits. In direct mapping the total main memory is accessed by few locations of cache memory. In order to implement the total address bits are classified as 4-bits for tag input and two for line input bits. The mathematical formulae for choosing tag and line bits are for n main memory locations and m cache memory locations.

- The number of line-in bit $s = \log_2 m$
- The number of tag-in bit $s = \log_2 n - \log_2 m$

Apart from tag, line and data bits the other signals from the CPU are read, write signals. Read signal acts as input signal to the de-multiplexer. The line-in bits are used as select lines for the de-multiplexers and multiplexers. For the read signal and line bits as select lines de-multiplexer acts as decodes to locate the required of block of memory, from the four locations. Out of the 4 output lines of de-multiplexer each line is given as enable to the memory cell of the cache memory. Each memory block of cache memory size of 12-bits of which 4-bits are tag bits and are data.

For every read and write signal of CPU tag of the data requested by the CPU and the tag of the cache memory are compared by the comparator, through proper switching by the de-

multiplexers. For the read cycle the comparator output '1' is said to be cache hit. A hit when talking about cache is when the processor finds data in the cache that it is looking for. A miss is when the processor looks for data in the cache, but the data is not available. In the event of a miss, the cache controller unit must gather the data from the main memory, which can cost more time for the processor. For cache hit comparator output is '1' which is given as input to the de-multiplexer. The de-multiplexer enables the block of data from the cache memory. For the same line bits the multiplexer of the output end produces the data from the cache memory to the output. A cache hit or miss is communicated with the CPU by a status signal of the comparator output.

In direct mapped cache read signal is given higher priority than the write signal i.e. when read signal is enabled write operation cannot be performed.

To implement this, a special function is developed at the write signal. A 2×1 multiplexer is used to prevent write operation during the read operation. Read signal from the CPU is used as select line for the mux, write is one of the input to the mux and the other input is logic '0'. So whenever read signal is enabled write signal is disabled by connecting to logic '0' and when read is disabled write is enabled connecting to the write of CPU.

During write cycle of the CPU, data is written to the cache, even if the data is present in the cache or not. The cache hit or miss is communicated with CPU by the status signal at the comparator output. When a write hit occurs copy of data is written to the cache and for a write miss data is written to the cache and the main memory.

3. COMPARATOR, MULTIPLEXER, DE-MULTIPLEXER, DATA REGISTERS, MEMORY CELL

Cache components include de-multiplexers and multiplexers for addressing the cache memory, comparators to compare the tag bits of the instruction from the processor with that in the cache memory and buffer register for storing the CPU instructions. Functionality and design of each block of cache memory is explained in the following sub sections

3.1 COMPARATOR

It is used to compare the tag bits of data requested by the CPU with the tag bits in the CPU. Here flag bits used are 4 so 4-bit binary identity comparator is used to compare the tag bits, whenever the tag bits are identical, comparator output is logic'1' as shown Fig.3.

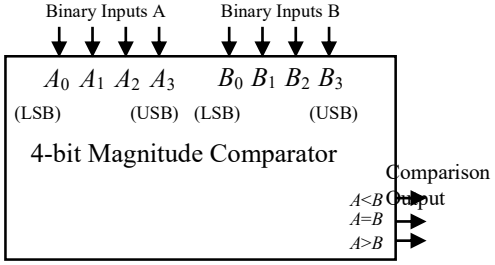

Fig.3. Comparator

3.2 MULTIPLEXER

Digital Multiplexers are constructed from individual analogue switches encased in a single IC package as opposed to the "mechanical" type selectors such as normal conventional switches and relays. Generally, multiplexers have an even number of data inputs, usually an even power of two, n^2 , a number of "control" inputs that correspond with the number of data inputs and according to the binary condition of these control inputs, the appropriate data input is connected directly to the output.

In cache memories multiplexers are used of the output section i.e. data from the cache memory for a cache hit is produced to the CPU by multiplexers 32×8 multiplexers is constructed by using eight 4×1 multiplexers.

3.3 DE-MULTIPLEXER

Unlike multiplexers which convert data from a single data line to multiple lines and de-multiplexers which convert multiple lines to a single data line, there are devices available which convert data to and from multiple lines. In cache memories de-multiplexers function as decoders. De-multiplexers address the blocks of cache memory. Select lines of de-multiplexers are taken from the line-in bits. Depending on the line-in bits, the respective block of cache memory is enabled.

3.4 DATA REGISTERS

Some registers do nothing more than storing a binary word. The buffer is the simplest of registers. It simply stores the binary word. The buffer may be a controlled buffer. Most of the buffer registers use D flip-flops. The binary to be stored is applied to the data terminals. On the input terminals, i.e. the input word is loaded into the register by the application of clock pulse.

3.5 MEMORY CELL

Stores all the cache memory data, read or write only. In the design, read is prominent, you cannot write while read is on, but you can read while write is on. The memory cells are designed using flip-flops, and modified to have two signals for read and write enables. In each memory line/cell we store 8-bits of actual data, and 4-bit for tag comparison.

4. IMPLEMENTATION

Cache memory implementation is done at various levels, i.e., the individual blocks of cache memory are designed at gate level. After the gate level implementation of the required block, flaws are detected early in the design cycle, allowing to make corrections with the least possible impact to schedule. This is more efficient for larger designs, for which synthesis and place and route can take a couple of hours.

The gate level implementation of 4 bit binary comparator is shown in Fig.4.

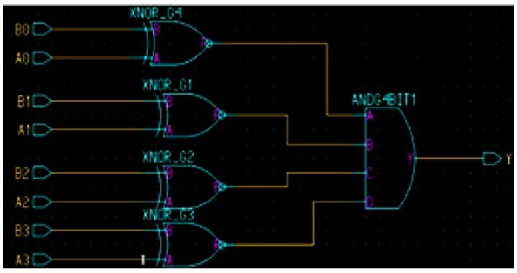

Fig.4. Gate level implementation of identity comparator

CMOS transistor level implementation of 4-input AND gate is shown in Fig.5, where *A*, *B*, *C* and *D* are inputs. Output is the AND operation of *A*, *B*, *C* and *D*.

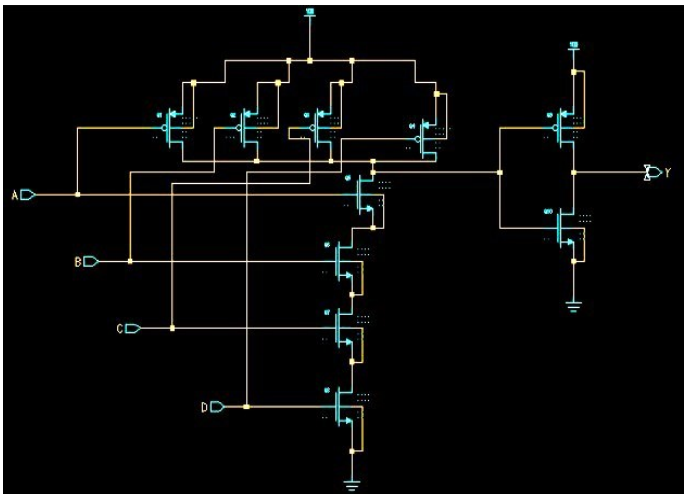

Fig.5. Transistor level implementation of 4bit and gate

CMOS transistor level implementation of 2-input XNOR gate is shown in Fig.6, where *A* and *B* are inputs. Output is the XNOR operation of *A* and *B*.

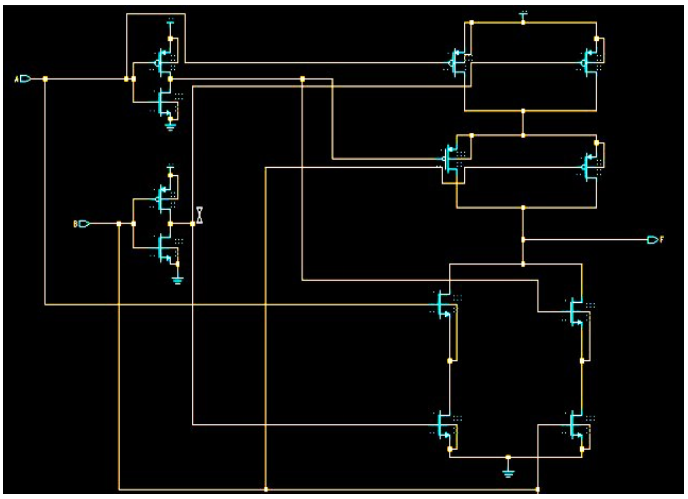

Fig.6. Transistor level implementation of XNOR gate

The gate level implementation of multiplexer is shown in Fig.7, where *A*, *B*, *C* and *D* are inputs and *a*, *b* are select lines and Q is the selected input.

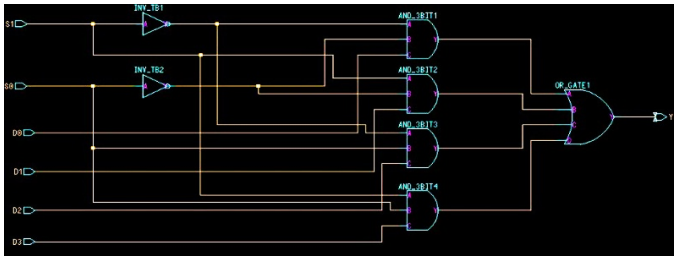

Fig.7. Gate level implementation of multiplexer

CMOS transistor level implementation of 4-input OR gate is shown in Fig.8, where *A*, *B*, *C* and *D* are inputs and *Y* is the logical OR operation of *A*, *B*, *C* and *D*.

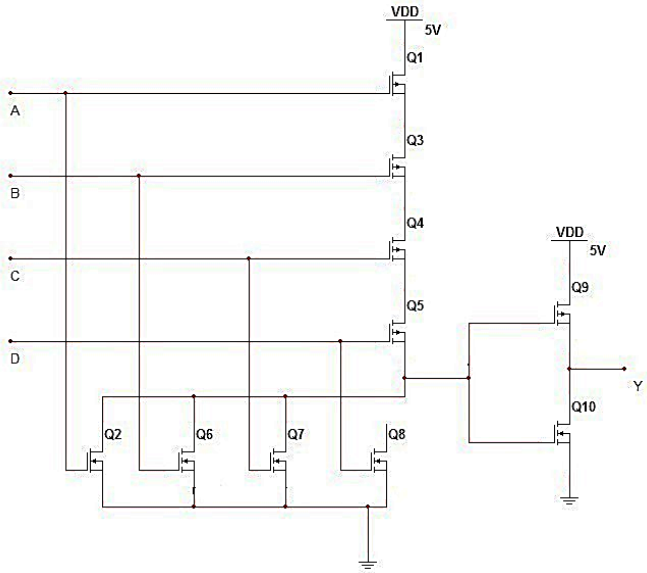

Fig.8. Transistor level implementation of 4-input OR gate

The gate level design implementation of De-multiplexer is shown in Fig.9, where *a*, *b* are select lines. *F* is the input and *A*, *B*, *C* and *D* are outputs of the de-multiplexer.

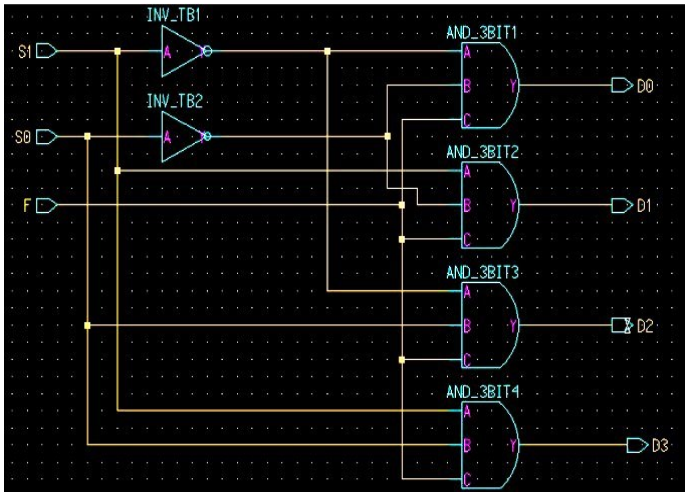

Fig.9. Gate level implementation of De-multiplexer

A 1-bit memory cell is constructed by using two D flip-flops connected in cascade as shown in Fig.10. Enable input of first flip-flop is used as write enable (*We*) and the enable (*Re*) of second

flip-flop is used as read enable for the memory cell. When write enable is active high, data can be written into the memory cell through the data port (*D*) of the D flip-flop.

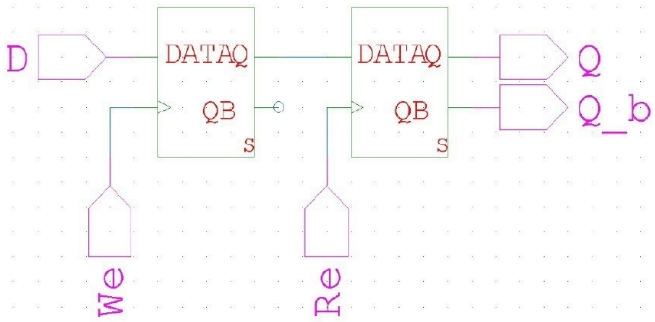

Fig.10. One BIT Memory Cell

When read enable is active high the data stored in the memory cell can be read through the port *Q*. In this paper, read and write enable signals are controlled by the de-multiplexers. D flip-flops are designed using NAND gates. First a NAND gate is designed and simulated at transistor level and it is used to design a d flip-flop as shown in Fig.11.

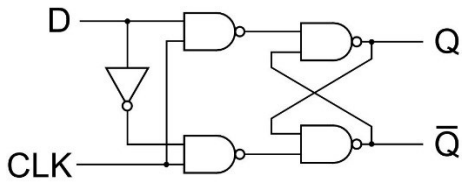

Fig.11. Gate level design of D flip-flop

The Fig.12 represents the 4-bit register constructed using 1-bit memory cells where *R*, *W* represent data read and write enables to the memory cell.

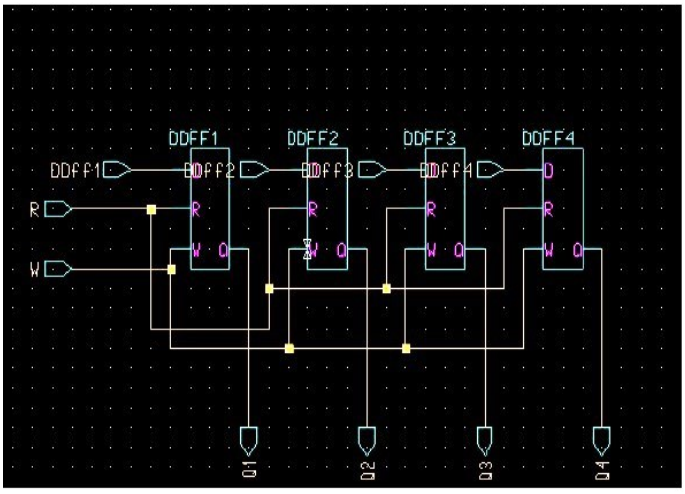

Fig.12. 4 bit register set constructed using 1-bit memory cell

Cache memory implementation is done at various levels, as shown in Fig.13 i.e. the individual blocks of cache memory are designed at gate level as shown in Fig.13. After the gate level implementation of the required block, flaws are detected early in the design cycle, allowing to make corrections with the least possible impact to schedule. This is more efficient for larger designs, for which synthesis and place and route can take a couple of hours.

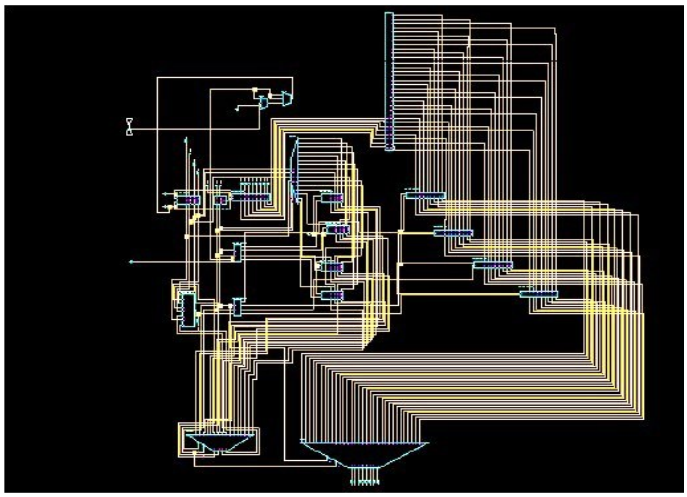

Fig.13. Gate level implementation of cache memory

When the designed circuit is flawless, each gate of the circuit is designed at CMOS transistor level in mentor graphics using design architect tool and symbols are generated accordingly. Each symbol generated is tested for various inputs to verify the circuit at CMOS level. These symbols after testing are further used to construct the main blocks of cache such as multiplexers, de-multiplexers, buffer registers and comparator.

5. RESULTS

When the designed circuit is flawless, each gate of the circuit is designed at CMOS transistor level in mentor graphics using Design architect tool and symbols are generated accordingly. Each symbol generated is tested for various inputs to verify the circuit at CMOS level. These symbols after testing are further used to construct the main blocks of cache such as multiplexers, de-multiplexers, buffer registers and comparator.

The Fig.14 represents the cache memory block implemented. Inputs to the cache include read (*R*), write (*W*), clock (*CLK*), enable (*EN*), tag, index inputs and data outputs to the cache include data out and status (*S*).

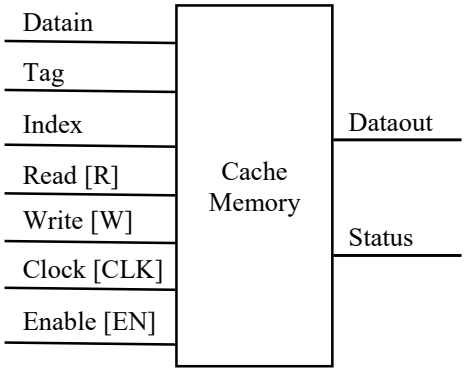

Fig.14. Cache memory inputs and outputs

When the designed blocks are tested, they are connected together to implement cache memory, then net lists are created

and activated by the generated inputs. Inputs to the cache memory are chosen to implement the read and write policies of cache.

Cache memory works based in the input signals from the processor, in this paper deals with the simulation of cache only, the inputs signals such as read, write and data are chosen and generated by us. For the given inputs the functionality of cache is observed and its performance is analyzed.

5.1 DESIGN BLOCKS OF CACHE AND THEIR OUTPUTS

The comparator block designed has the inputs A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3 where A, B are binary 4 bit numbers. The output of the identity comparator is given by Y as shown Fig.15.

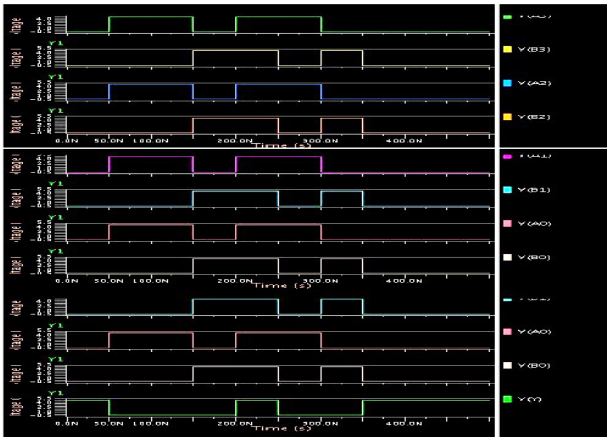

Fig.15. Input and output wave forms of the comparator

The multiplexers used in 4×1 and 2×1 multiplexers. 4×1 multiplexers are further used to build 8×32 and 4×16 multiplexers. The signals to the multiplexers are select lines S_0, S_1 and input signals I_0, I_1, I_2, I_3 here I_0 is made high and others are made low as shown in Fig.16.

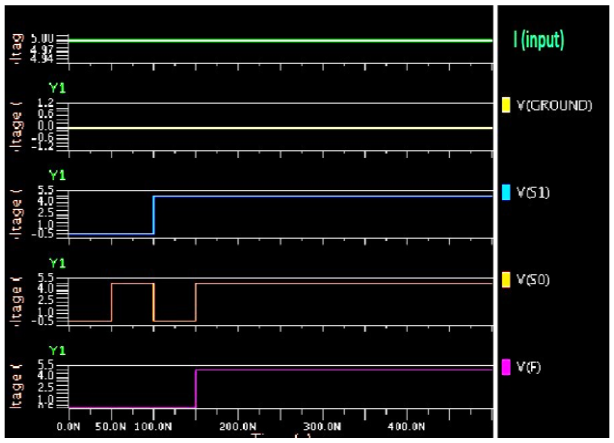

Fig.16. Input and output wave forms of the multiplexer

In a 1×4 de-multiplexer I is the input, S_0, S_1 are the select lines and D_0, D_1, D_2, and D_3 are outputs of the de-multiplexer as shown in Fig.17.

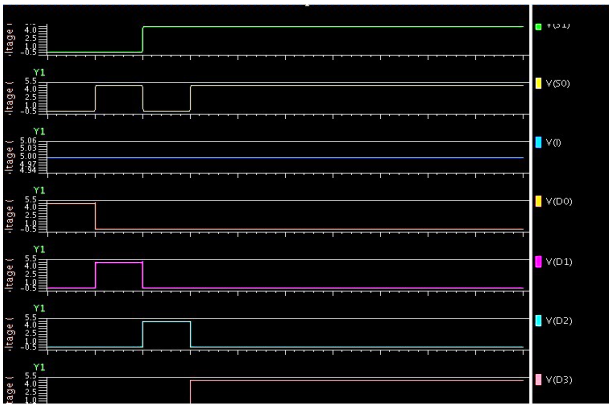

Fig.17. Input and output wave forms of the De-multiplexer

Registers of cache memory are group of 1-bit memory cells. Two D-flip flops connected together in cascade are used as 1-bit memory cell. Each D-flip flop is designed using NAND gates. DFF1, DFF2 are the data inputs to the registers and Q_1, Q_2 are outputs of register as shown in Fig.18.

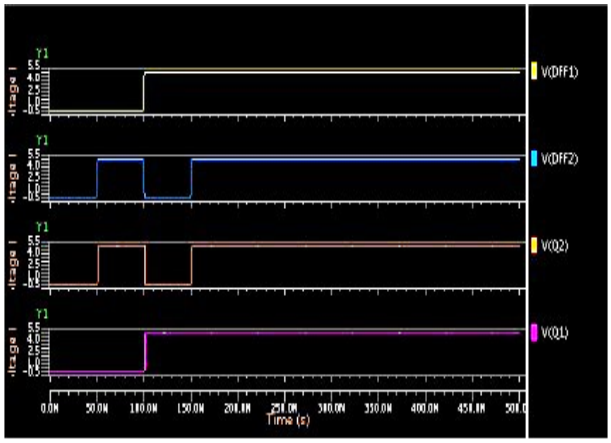

Fig.18. Input and output waveforms to the D flip-flop

The Fig.19 represents the 8-bit data buffer register constructed using 1-bit memory cells.

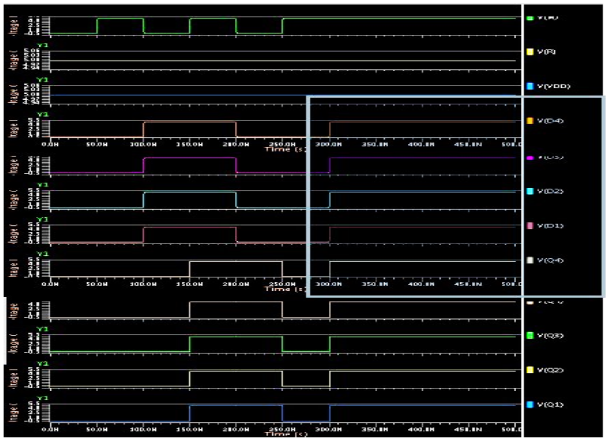

Fig.19. Input and output wave forms to the 8-bit data register

The Fig.20 represents the CPU inputs assigned to the cache memory.

Fig.20. CPU inputs to the cache memory

The Fig.21 represents the data inputs given to the cache memory.

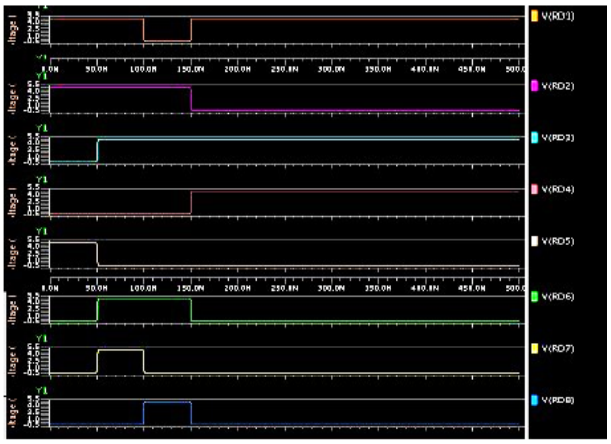

Fig.21. Data inputs to the cache memory

The Fig.22 represents the tag and index inputs given to the buffer registers of the cache memory. Tag and index data are chosen such that for a read signal from the CPU a cache hit occurs. Similarly for every write signal enable cache write occurs.

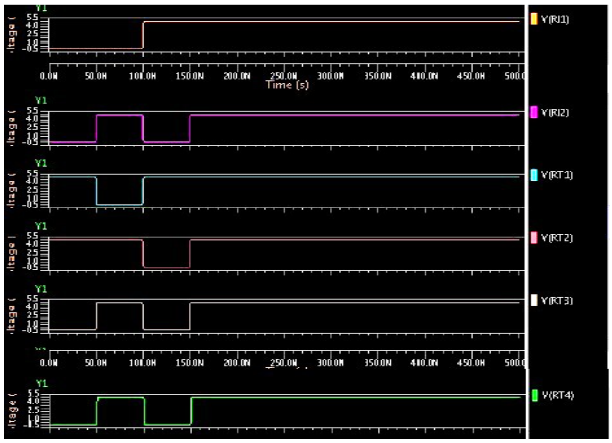

Fig.23. Tag and Index input to the cache memory

The Fig.24 represents the cache outputs, data out and status for the given CPU inputs. Results of the cache were satisfactory, implementing the functionality of the cache memory.

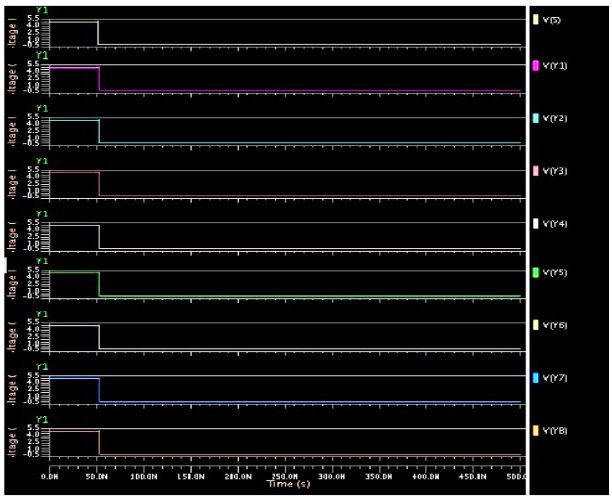

Fig.24. Status and output data of cache memory

5.2 PERFORMANCE OF CACHE

Performance of cache is evaluated based on the power dissipation and speed of operation. The Table.1 gives the information about various block with respect to nodes, elements and input signals.

Table.1. Comparison of Various Blocks

Component Name	Nodes	Elements	Input Signal
Register tag	79	151	4
Register index	22	151	4
Register data	38	67	9
Comparator	24	44	4
De-multiplexer	29	54	4
Multiplexer	12	23	3
Cache Memory	1738	3454	20

The Table.2 gives the data about the total time taken by cache and its components for simulation.

Table.2. Total time taken by cache for simulation

Component Name	CPU			
	5V	3V	1.8V	1V
Register tag	150ms	120ms	110ms	110ms
Register index	80ms	70ms	60ms	70ms
Register data	480ms	440ms	280ms	340ms
Comparator	70ms	80ms	70ms	60ms
De-multiplexer	20ms	20ms	30ms	20ms
Multiplexer	20ms	30ms	20ms	20ms
Cache Memory	7.3ms	6.2ms	5.4ms	5.6ms

The Table.3 gives the information about the power dissipation of cache and its components for various voltage levels of operation.

Table.3. Comparison of various blocks in terms of power dissipation

Component Name	Power Dissipation at Various Voltage Levels			
	5V (µW)	3V (µW)	1.8V (µW)	1V (µW)
Register tag	34.662	87.9861	1.6652	136.710
Register index	5.7772	0.7055	290.91	28.1400
Register data	46.411	8.2749	20.490	9.1100
Comparator	5.6391	14.844	345.10	45.7100
De-multiplexer	4.3601	11.074	235.86	25.9023
Multiplexer	7.2667	18.396	369.90	35.8900
Cache Memory	676.55	1.5843	30.80	17.7700

6. CONCLUSIONS

Cache Memory has been successfully designed and implemented. All verification data appears to meet the design criteria. There were unpredictable design errors on the way, but none of that stopped the cache memory to function normally. In order to overcome the errors the actual design of cache has been modified accordingly. The performance of cache is evaluated, in terms of power dissipation and speed of operation.

This design can be expanded to include fetching control systems to a Main Memory system. This is a functionality that can be added on in the future. The project can be further expanded to improve the cache size, to determine the maximum size of cache that is possible using the type of memory cells that we have. Other improvements would be to actually use 6T SRAM cell design for the memory cell instead of flip-flops that requires more area due to more transistors in each memory cell.

REFERENCES

[1] John L Henessey and David A Petterson, "*Computer Architecture: A Qualitative Approach*", 5th Edition, Morgan Kaufmann, 2011.

[2] J.H. Chang, H. Chao and Kimming So, "Cache Design of a Sub-Micron CMOS System/370", *Proceedings of 14th Annual International Symposium on Computer Architecture*, pp. 208-213, 1987.

[3] Jongsok Choi, Kevin Nam, Andrew Canis, Jason Anderson, Stephen Brown and Tomasz Czajkowski, "Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems", *Proceedings of IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 241-248, 2012.

[4] John P. Hayes, "*Computer Architecture and Organization*", Mc Graw Hill, 1988.

[5] N.P. Jouppi, "Cache Write Policies and Performance", *Proceedings of 20th Annual International Symposium on Computer Architecture*, pp. 1911-1920, 1993.

[6] A. Milenkovic, M. Milenkovic and N. Barnes, "A Performance Evaluation of Memory Hierarchy in Embedded Systems", *Proceedings of 35th Southeastern Symposium System Theory*, pp. 427-431. 2003.

[7] T.S. Warrier, B. Anupama and M. Mutyam, "An Application-Aware Cache Replacement Policy for Last-Level Caches", *Proceedings of International Conference on Architecture of Computing Systems*, pp. 207-219, 2013.

[8] Jianwei Dai and L. Wang, "An Energy-Efficient L2 Cache Architecture using Way Tag Information under Write-Through Policy", *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 21, No. 1, pp. 102-112, 2012.

[9] J. Hong, J. Kim and S. Kim "Exploiting Same Tag Bits to Improve the Reliability of the Cache Memories", *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 23, No. 2, pp. 1-7, 2014.

[10] Vipin S. Bhure and Praveen R. Chakole, "Design of Cache Controller for Multi-core Processor System", *International Journal of Electronics and Computer Science Engineering*, Vol. 1, No. 2, pp. 520-527, 2013.

[11] H. Noguchi et al., "Highly Reliable and Low-Power Nonvolatile Cache Memory with Advanced Perpendicular STT-MRAM for High Performance CPU", *Proceedings of Symposium on VLSI Circuits Digest of Technical Papers*, pp. 1-2, 2014.